ELSEVIER

Theoretical Computer Science 233 (2000) 247–261

Theoretical Computer Science

www.elsevier.com/locate/tcs

Tight bounds on pebbling with faults [1]

Yonatan Aumann [a,*,2], Judit Bar-Ilan [b,3], Uriel Feige [c,4]

[a] *Department of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, Israel*
[b] *School of Library, Archive and Information Studies, The Hebrew University Jerusalem 91904, Israel*
[c] *Department of Applied Math., The Weizmann Institute, Rehovot, Israel*

Received September 1996; revised February 1998
Communicated by D. Peleg

Abstract

We introduce a formal framework to study the time and space complexity of computing with faulty memory. For the fault-free case, time and space complexities were studied using the "pebbling game" model. We extend this model to the faulty case, where the content of memory cells may be erased. The model captures notions such as "check points" (keeping multiple copies of intermediate results), and "recovery" (partial recomputing in the case of failure). Using this model, we derive tight bounds on the time and/or space overhead inflicted by faults. As a lower bound, we exhibit cases where f worst-case faults may necessitate an $\Omega(f)$ multiplicative factor overhead in computation resources (time, space, or their product). The lower bound holds regardless of the computing and recomputing strategy employed. A matching upper-bound algorithm establishes that an $O(f)$ multiplicative overhead always suffices. For the special class of *binary tree computations*, we show that f faults necessitates only $\Theta(f)$ *additive* factor in space.

Keywords: Fault tolerance; Pebbling; Pebbling games

1. Introduction

We study the time and space complexities of computing in an environment with faulty memory. Consider a large-scale computation task. In the course of the computation, many intermediate results are computed and stored in memory for later reference.

* Corresponding author.
E-mail address: aumann@cs.biu.ac.il (Y. Aumann)

[1] A preliminary version was presented in ICALP '94.

[2] This work was done while the author was at the Weizmann Institute, Rehovot, Israel.

[3] Part of this work was carried out while the author was with the Department of Applied Math., The Weizmann Institute of Science.

[4] Work done while author was supported by a Koret Foundation fellowship.

PII: S0304-3975(98)00085-1

Memory, however, is not fully secure, and at times the content of a memory unit may be lost (due to hardware failure, power failure, cosmic radiation, etc.). In this case, the computation must recover the corresponding data from elsewhere, possibly by retracing previous computations. What is the overhead introduced by this recomputing? Is there an alternate order to perform the computation which will accelerate recovery? We may also consider giving additional space to the program, in order to facilitate quick recovery. For example, essential data may be duplicated in memory, thus decreasing the chances of losing it. A standard practice is to place periodic "check points" along the computation, down-loading a full copy of the entire memory. Later, the computation need only be recovered from this check point. Note, however, that these duplicate copies may themselves be subject to failures. Whatever the method may be, can additional memory considerably decrease the recomputing time? How much additional space is necessary for these techniques to be effective?

We present a formal framework to study these questions. Previously, time-space complexity of straight-line programs was studied using the "pebbling game" model [1–4, 8]. We extend this model to the faulty case, allowing memory units (e.g. registers, disc sectors) to be erased during the computation. The faulty scenario is modeled as a *two person game*, with a *computing player* aiming to complete the computation with the space provided to him, and an *erasing adversary player* slowing him down by erasing memory units. The model assumes that all memory units are alike, and that when the content of a unit is erased, this is evident to the program. The model captures notions such as check-points and recomputing. We do not, however, allow for more sophisticated mechanisms such as error-correcting and alike.

Using this model, we establish tight bounds on the time-space complexity of the computation in the presence of faults. The main result is a negative one: there exist computations for which recovery from f faults requires an $\Omega(f)$ multiplicative factor overhead in the resources of the system (time, space, or their product). Specifically, given a k factor additional space, the computation may still take $\Omega(f/k)$ times longer. In particular, a computation which without faults completes with s space, when faced with f faults may necessitate as much as $\Omega(fs)$ space in order to keep the computation time linear. The lower bound holds regardless of the computing strategy used for the computation, and even if the program knows in advance the number of faults and is informed of all failures immediately as they occur. We note that the computations which exhibit this worst case performance are of a rather simple structure, and similar computations are frequently carried out in practice (e.g. FFT computations).

Matching to the lower bound, we present an effective algorithm which achieves the same asymptotic performance, for any computation. In essence, the algorithm calls for keeping "snap-shots" of the entire memory at fixed time intervals during the computation. These snap-shots facilitate recovery in case of faults.

For some computations, better performance can be obtained. We exemplify this by considering computations which have a tree-like structure. For this class of computations, we provide an algorithm which only necessitates an $O(f)$ *additive* factor in space

and constant multiplicative factor in time in order to overcome f faults. A matching lower bound for this class is proven.

As we have previously stated, our main result is a negative one. We show that if we only allow to store *duplicates* of the intermediate results, and faults are to be overcome by recomputing, then the overhead may be excessive. While we present a matching algorithm, we do not necessarily advocate using it. Our results may be interpreted to indicate that stronger and more sophisticated mechanisms should be employed. In particular, a strong alternative is using *error-correcting codes* to periodically encode the full contents of memory and store it in an auxiliary space. This only incurs a linear overhead in space requirements and guarantees that the state of the memory can later be reconstructed even if the number of faults is considerable. Initially, the encoding procedure is time consuming, but our results indicate that if faults are to be expected, it may well be worth while. A particular form of error correcting codes, which is more suitable for this setting is the *Information Dispersal Algorithm* (IDA) [6, 7]. IDA allows for efficient reconstruction in the case of erasures, and is relatively efficient in the encoding and decoding procedures.

This paper is organized as follows. In the next section (Section 2), we present the pebbling game model for the faulty case, and define the complexity measures. In Section 3 the lower bound theorem is stated and proved. The matching upper bound algorithm is given in Section 4. The results for the special class of tree computations are presented in Section 5.

2. The pebbling game

Pebbling has been used to study memory requirements of straight line programs, to study flowcharts, and to derive time-space tradeoffs (see [1–4, 8] and references therein). We extend the model to the faulty case, modeling the case where the content of memory units may be erased.

The *faulty pebbling game* is defined as a two person game, played on a directed acyclic graph G with bounded fan-in. The graph G represents the computation to be performed. The *computing player*, denoted by C, is given a set of pebbles and seeks to pebble all nodes of the graph. The game is played in discrete time (steps). Pebbles may be placed on the nodes of the graph according to the following rules:

1. For a node w, denote by $\mathrm{Pre}(w) = \{u : (u, w) \in E\}$, the immediate predecessors of w. If at time t all nodes of $\mathrm{Pre}(w)$ have a pebble placed on them, then at time $t+1$ a pebble can be placed on w (in particular, a pebble can always be placed on an input node).
2. If a node holds a pebble, then another pebble can be placed on the same node.
3. Only one pebble can be placed in each time step.
4. At the end of each time step, the computing player may remove a pebble from an arbitrary node (so as to have a free pebble available for the next time step).

Intuitively, a node w corresponds to a (partial) computation value, which can be computed as a function of the values in the predecessors. Pebbles correspond to memory units. Placing a pebble on a node means that this memory unit holds the value corresponding to the node. Removing a pebble from a node means that the computing player no longer relies on the memory unit to store the value corresponding to the node, and the memory unit can find other future use. The objective of player C is to pebble all nodes of the graph (not necessarily simultaneously).

Alternating with C, the *erase player*, denoted by E, makes his moves. Following each pebbling step of C, player E may remove any number of pebbles from the graph, and from any location. These pebbles are given back to C for future use. Intuitively, removing a pebble corresponds to erasing the content of the corresponding memory unit. We place a bound f on the total number of pebbles that E can remove throughout the game. We thus obtain the following definition.

Definition 1. A *faulty pebbling game* is a triplet (G, s, f), where G is a directed acyclic graph with bounded fan-in, s the number of pebbles given to C, and f the bound on the number of pebbles E may remove.

We note that when $f = 0$ (no faults) our game essentially reduces to the standard one [1]. The only additional rule is Rule 2, which is superfluous if no faults occur.

Given a game (G, s, f) the objective of player C is to minimize the number of steps necessary to complete the pebbling of G, and player E the opposite. We often refer to E as the *adversary*. For given strategies $\mathscr{C}, \mathscr{E}$, of C and E respectively, $T_G(s, f, \mathscr{C}, \mathscr{E})$ is the total pebbling time of G. We define

$$T_G(s, f) \stackrel{\text{def}}{=} \max_{\mathscr{E}} \min_{\mathscr{C}} T_G(s, f, \mathscr{C}, \mathscr{E}) = \min_{\mathscr{C}} \max_{\mathscr{E}} T_G(s, f, \mathscr{C}, \mathscr{E}) \ .$$

The second equality follows from the Min–max Theorem of Game Theory [5]. Note that this is a full information game. Thus, from the same Min–max Theorem it also follows that this value for $T_G(s, f)$ can be obtained with the players using only deterministic strategies (*pure strategies*), without employing randomization.

In this work we are interested to determine the extra time and/or space required for computing in the faulty setting. Thus, we consider the relationship between $T_G(s, 0)$, the computing time with no faults, and $T_G(\hat{s}, f)$, the computing time with f failures and $\hat{s} \geqslant s$ pebbles.We consider a large range of possible values for these parameters and derive the trade-offs.

2.1. An example

Consider the process of updating a variable by performing a loop n times. The DAG corresponding to this process is a line of $n+1$ nodes, v_0 to v_n, and n edges, (v_i, v_{i+1}). v_0 corresponds to the initial value of the variable, and v_i corresponds to the value of the variable after i iterations of the loop. In our model, two memory units (pebbles) are necessary and sufficient in order to carry out the computation. One memory unit holds

the current contents of the variable. The other is used to store the updated value after a new execution of the loop. Thereafter, the first memory unit, which holds outdated information, can be freed and made available to store future values of the variable. In terms of pebbling operations, initially we place a pebble on v_0, signifying the fact that the initial value of the variable is known. At any time step, if there is a pebble on node v_i, we may place a pebble on node v_{i+1}, and remove the pebble from v_i. The computation process ends when v_n is pebbled. Altogether, the number of pebbles used is 2, and the number of steps (including initialization) is $n+1$.

If after some time step (e.g., when there is one pebble free and the other on v_i) a memory fault occurs, the pebble is removed from v_i (that is – the value of the variable is erased). Now the whole computation has to be redone. A pebble is placed on v_0 (we assume that the initial value of the variable is an external input to the computing player, and hence can be reconstructed), and all nodes are repebbled one by one.

Evidently, f successive faults, each occurring just after v_{n-1} was pebbled (and the pebble from v_{n-2} removed), cause the most harm, increasing the time until v_n is pebbled by an additive factor of fn (or a multiplicative factor of nearly $f+1$).

If the computing player has $\hat{s} > 2$ pebbles, then the effect of memory faults can be reduced. We sketch how this is done. The computing player may allocate two of the pebbles in order to perform the usual fault free pebbling of the graph, and use the remaining $p = \hat{s} - 2$ pebbles to take "snapshots" of intermediate steps of the computation. These pebbles are placed on nodes $v_{i/(p+1)}$, for $1 \leqslant i \leqslant p$, once each of these nodes is reached by the pebbling process. Now any pebble that is removed by the adversary can be reconstructed from previous pebbles. If several pebbles are removed, then the lowest pebble is reconstructed first. The adversary can also remove pebbles during reconstruction. In this case, the reconstruction of a pebble is restarted. In all, each fault can result either in the reconstruction of a single "snapshot" pebble, or in redoing work performed in between "snapshots". In both cases, the extra work is at most $O(n/p)$. Hence, f faults cause a time overhead of only $O(fn/p)$, or a constant multiplicative overhead in time if $p \simeq f$.

The example of the line indicates that for some DAGs, in order to restrict the effect of f faults to a constant multiplicative overhead in pebbling time, one needs to multiply the number of pebbles by a factor of f. Indeed, we prove this (for a different DAG) in Section 3. The example of the line falls short of proving this, because the number of pebbles used in the faultless case is only constant, and hence there is no distinction between multiplicative and additive overhead in the number of pebbles in the faulty case. In fact, as Section 5 shows, faults in binary trees can be handled with only an additive factor of $O(f)$ pebbles, and the line is a special case of a binary tree.

3. The lower bound

In this section we prove the main lower bound theorem.

Theorem 1. *For any T and s, there exists a graph $G = G_{T,s}$, such that $T_G(s,0) = T$ and for all $f \leqslant T/s$ and $s \leqslant \hat{s} \leqslant T/39 \log s$,*

$$\frac{\hat{s} \cdot T_G(\hat{s}, f)}{s \cdot T_G(s,0)} = \Omega(f).$$

Thus, the overhead due to failures is a linear multiplicative factor in the resources of the system, time, space, or their product.

We now proceed to prove the theorem. First we present the construction of the graph $G_{T,s}$ and prove several properties regarding the pebbling of this graph. Then we provide the adversary's strategy. Finally, the time bound is proven.

3.1. Structure of $G_{T,s}$

The graph $G_{T,s}$ is constructed as a sequence of butterfly graphs and then a sequence of trees. Each butterfly consists of $\Theta(s \log s)$ nodes ($\Theta(s)$ inputs and outputs). The full sequence consists of $\Theta(T/s \log s)$ butterflies. Following this sequence, we attach a sequence of full binary trees, each with $\Theta(s)$ inputs. The root of the one tree is connected to *all* the inputs of the next tree. In addition, we draw an edge form each input node of the tree to the corresponding input node of the next tree. The full sequence is composed of $\Theta(T/s)$ such trees. Finally, we add a chain of nodes starting at the root of the last tree, to obtain exactly T nodes in the graph. We now proceed to give a formal description of the graph.

For $x \in \{0,1\}^k, x = x_0 x_1 \ldots x_{k-1}$, denote by x^i the string $y \in \{0,1\}^k$ which differs from x in the ith coordinate alone. The *k-butterfly* is the graph $B = (V,E)$, with

$$V = \left\{(i,x) : i \in \{0,\ldots,k\}, x \in \{0,1\}^k\right\}$$

and

$$E = \{((i,x),(i+1,x)) : i \in \{0,\ldots,k-1\}\} \cup \{((i,x),(i+1,x^i)) : i \in \{0,\ldots,k-1\}\}.$$

A 3-butterfly is depicted in Fig. 1.

Choose N, $s/4 < N \leqslant s/2$ such that N is a power of 2. Let $k = \log N$, and set $m = \lfloor T/3kN \rfloor$ (for our parameters, $m > 1$). The first part of the graph is composed of a sequence of m butterflies, $B_1,\ldots,B_m$. Each butterfly B_i is a k–butterfly (N inputs and N outputs). We identify the inputs of the B_{i+1} with the outputs of B_i ($i = 1,\ldots,m-1$). Following the sequence of butterflies we attach a sequence of $k \cdot m$ full binary trees, each with N inputs and depth k. The edges are directed form the leaves to the root. Denote these trees by $C_1,\ldots,C_{km}$. The inputs (=leaves) of C_1 are identified with the outputs of B_m. For tree C_i denote by $z_j^{(i)}$ the jth leaf of C_i and its root by $root^{(i)}$. For $i = 1,\ldots,km-1$, and each j we draw a directed edge from $z_j^{(i)}$ to $z_j^{(i+1)}$. In addition, we place edges from $root^{(i)}$ to *all* inputs of C_{i+1}. Let K be the number of nodes in the graph so far. To complete the graph to size T we add a chain of nodes of length

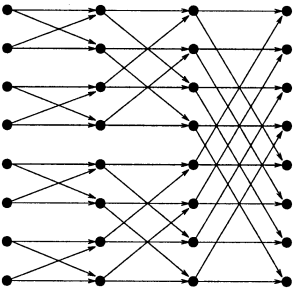

Fig. 1. A 3-butterfly.

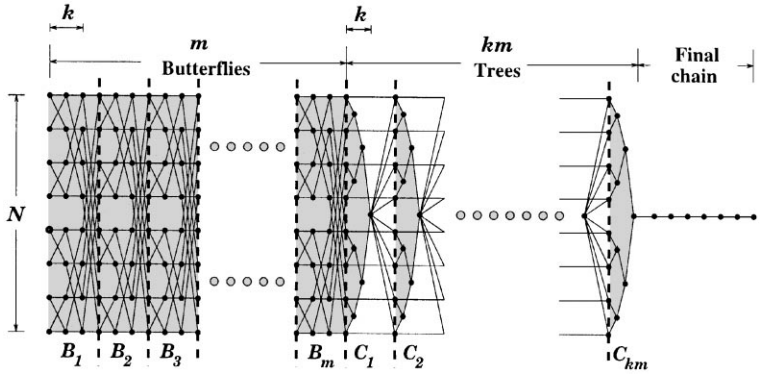

Fig. 2. A schematic picture of $G_{T,s}$ with $k = 3$, $N = 8$.

$(T - K)$ following the root of the last tree. A schematic picture of the entire graph $G_{T,S}$ is depicted in Fig. 2. Let $G = G_{T,s}$.

Claim 1. $T_G(s, 0) = T$.

Proof. By construction $|G| = T$. The butterflies can be pebbled with $N + 2$ pebbles, column after column. We use the property that in every column, nodes can be paired in such a way that each pair of nodes in the column effects only two nodes in the next column. Hence, if N pebbles are used to pebble a column, two extra pebbles (together with the existing N pebbles) suffice in order to pebble the next column. The trees can be pebbled with $N + \log N + 2$ pebbles. We leave N pebbles on the leaves of a tree (since they are needed in order to pebble the leaves of the next tree), and use the remaining $\log N + 2$ pebbles to get the root of the tree pebbled. (To see that $\log N + 2$ pebbles suffice, one may use induction on the depth of the tree. A formal proof is provided in Lemma 3.) The final chain can be pebbled with two pebbles (as in Section 2.1). No node is ever pebbled twice. By construction $N + \log N + 2 \leqslant s$. □

Definition 2. Let H be a graph, and consider a given placement of pebbles on H. Let $\mathscr{P}$ be a directed path in H, and let w be the last node of $\mathscr{P}$. We say that $\mathscr{P}$ is *d-pebbled* if there are d pebbles placed on the nodes of $\mathscr{P} - \{w\}$.

Definition 3. Let B be a k-butterfly ($k = \log N$) with ℓ pebbles placed on it. An output node w is *weak* if there exists a set of inputs I_w, called the *easy* set of w, such that
1. $|I_w| > N/2$.
2. For each $u \in I_w$, the path from u to w is less than $4\ell/N$-pebbled.
The set of weak nodes is called the weak end of B.

Claim 2. *Any butterfly B has at least $N/2$ weak outputs.*

Proof. For each pair u, w, with u an input and w an output, there is exactly one directed path from u to w. Each node of B is contained in exactly N such paths. Thus, each pebble placed on B is situated on exactly N input–output paths. Suppose the contrary to the claim. Then there is a set $\bar{O}$ of $(N/2+1)$ outputs, such that for each $w \in \bar{O}$ there is a set $\bar{I}_w$ of at least $N/2$ inputs such that for each $u \in \bar{I}_w$ the path from u to w is (at least) $4\ell/N$-pebbled. Counting multiplicity, this accounts for $N\ell + 2\ell$ pebbles. Each pebble is counted at most N times. Thus, there must be more than ℓ pebbles on B, in contradiction. □

Claim 3. *Consider a given placement of pebbles on G. Let ℓ_{tree} be the number of pebbles placed on the entire set of trees in G. For $j = 1, \ldots, N$, let L_j be the straight path $L_j = (z_j^{(1)}, z_j^{(2)}, \ldots, z_j^{(m)})$. At least $N/2 + 1$ of paths L_j have at most $2\ell_{tree}/N$ pebbles on them.*

Proof. Counting. □

Accordingly, we call these paths, *light straight paths*.

3.2. The adversary's strategy

We now present the strategy the adversary employs for removing pebbles. Recall that $\hat{s}$ pebbles are given to the pebbling player, and the adversary may remove at most f pebbles. For each tree $C_1, C_2, \ldots$, the adversary plays immediately following the first time a pebble is placed on the tree, and at these times only. At each such time the adversary removes at most $O(\hat{s}/s)$ pebbles but will necessitate an additional $\Omega(T)$ pebbling steps before the first pebble can be placed on the next tree. The adversary keeps doing so for as long as it can without exceeding the f faults limit.

Consider a time point t_i, and suppose that t_i is the first time that a leaf of C_i was pebbled. We now describe how the adversary chooses the pebbles to remove. We construct a path, which we call *the weak path*, which starts at an input node of G, runs through the *weak ends* of all butterflies, continues by a *light straight path* through the sequence of trees, and terminates at $root^{(i)}$ of C_i. We denote this path by $\mathscr{W}_i$. The

adversary removes all pebbles placed on the weak path. The path we construct will be at most $O(\hat{s}/s)$-pebbled.

For each $i = 1, \ldots, m$, let O_i be the weak-end of B_i. By Claim 2, $|O_i| \geqslant N/2$. Consider O_m (the weak-end of the last butterfly) and the straight paths L_j's emerging from O_m. By Claim 3, at least $N/2 + 1$ of the staight paths are light. Thus, there must be at least one light staight path L_{j_0} starting at a node of O_m. Set $w_m = z_{j_0}^{(1)}$, the input node (in O_m) of L_{j_0}, and $w_{m+1} = z_{j_0}^{(i)}$ the leaf node where C_i intersects L_{j_0}. Assume w_{j+1} has been determined, we show how to determine w_j. By induction assume that $w_{j+1} \in O_{j+1}$. Node w_{j+1} is weak. Let $I_{w_{j+1}}$ be the easy set (in B_{j+1}) of w_{j+1} (as in Definition 3). Thus, $|I_{w_{j+1}}| > N/2$. The inputs of B_{j+1} are identified with the outputs of B_j. Thus, $O_j \cap I_{w_{j+1}} \neq \emptyset$. Choose w_j to be any node in this intersection. For the case $j = 0$ choose w_0 to be any node of I_{w_1}. The *weak path* $\mathscr{W}_i$, is defined to be the path connecting w_0 to w_1 to w_2 etc. up to w_m, and then along the corresponding L_j to w_{m+1}, and from there to $root^{(i)}$.

Claim 4. *The weak path is at most* $4\hat{s}/N$*-pebbled.*

Proof. Let ℓ_{tree} be the total number of pebbles placed on the trees of G. By construction, the chosen straight path L_j is at most $2\ell_{tree}/N$-pebbled. For each j, let ℓ_j be the number of pebbles placed on nodes of B_j other than its output nodes (which are counted as input nodes of B_{j+1}, or of the first tree). By definition, the path from w_j to w_{j+1} is less than $4\ell_j/N$-pebbled. By assumption $\sum_{j=1}^{m} \ell_j + \ell_{tree} \leqslant \hat{s}$. Thus, the total number of pebbles on the weak path is

$$\leqslant \sum_{j=1}^{m} 4\frac{\ell_j}{N} + 2\frac{\ell_{tree}}{N} \leqslant \frac{1}{N}\left[4\sum_{j=1}^{m}\ell_j + 2\ell_{tree}\right] \leqslant 4\frac{\hat{s}}{N}\,. \qquad \square$$

Thus, the strategy that the adversary employs is that immediately following the first time a pebble is placed on C_i, it removes all pebbles placed on $\mathscr{W}_i$. We call this act *de-pebbling* of $\mathscr{W}_i$. The adversary continues to de-pebble as long as it can without exceeding the limit f of removals.

3.3. Pebbling time

Consider a given placement of pebbles on G. Denote this placement by D. For nodes $w, u \in G$ say that w *is reachable from* u if there exists a directed path from u to w which is 0-pebbled. Denote

$$R_D(w) = \{u : w \text{ is reachable from } u\}.$$

Lemma 1. *For any pebble placement* D *and node* w*, all nodes of* $R_D(w)$ *must be pebbled before* w *can be pebbled.*

Proof. By induction on the length of the path from u to w. $\square$

Claim 5. *Let t be the first time step when a pebble was placed on C_i. Suppose the adversary now de-pebbles $\mathscr{W}_i$. Then, at least $T/12 - 3k\hat{s}$ pebbles must be placed before $root^{(i)}$ can be pebbled.*

Proof. Let D be the pebble placement following the de-pebbling. We count the number of nodes in $R_D(root^{(i)})$. Consider the placement of the pebbles on the butterflies of G. Say that a butterfly B_j is *light* if at most $N/4$ pebbles are placed on it, and *heavy* otherwise. By a counting argument, at most $4\hat{s}/N$ butterflies are heavy. Say that a butterfly is *fragile* if it is light and the butterfly following it is also light. There are at least $m - 8\hat{s}/N - 1$ fragile butterflies (each heavy butterfly prevents two butterflies from being fragile, and in addition, B_m, which has no butterfly following it, cannot be fragile). We prove that in each fragile butterfly there are at least $kN/4$ nodes which are in $R_D(root^{(i)})$.

Let B_j be a fragile butterfly. Consider the node $w_{j+1} \in \mathscr{W}_i$. By construction, w_{j+1} is a weak output node of B_{j+1}, and B_{j+1} is light. Let $I_{w_{j+1}}$ be the easy set of w_{j+1} (Definition 3). By definition, for each $u \in I_{w_{j+1}}$ the path from u to w_{j+1} is less than $4(N/4)/N$-pebbled (since, B_{j+1} is light, we have $\ell \leqslant (N/4)$ is Definition 3). Thus, all these paths are not pebbled at all. Thus, $R_D(w_{j+1}) \supseteq \bigcup_{u \in I_{w_{j+1}}} R_D(u)$. The nodes of $I_{w_{j+1}}$ are also output nodes of B_j. Consider the straight-edge paths connecting these nodes with the corresponding input nodes of B_j (i.e. from a tth input to the tth output). Each such path is of length k and the paths are disjoint. Since B_j is light, at most $N/4$ of these paths contain a pebble. For any unpebbled path, the entire path is fully contained in $R_D(w_{j+1})$. Thus,

$$|R_D(w_{j+1}) \cap (B_j - B_{j+1})| \geqslant \left| \bigcup_{u \in I_{w_{j+1}}} R_D(u) \cap (B_j - B_{j+1}) \right| \geqslant \frac{kN}{4} .$$

However,

$$R_D(root^{(i)}) \supset \bigcup_{w \in \mathscr{W}_i} R_D(w) \supseteq \bigcup_{j=1}^{m} R_D(w_j) .$$

Thus,

$$\begin{aligned} \left|R_D(root^{(i)})\right| &\geqslant \left(m - 8\frac{\hat{s}}{N} - 1\right)\frac{kN}{4} \\ &> \left(\frac{T}{3kN} - 1 - \frac{8\hat{s}}{N} - 1\right)\frac{kN}{4} > \frac{T}{12} - 3k\hat{s} . \end{aligned}$$

□

We are now ready to prove the main theorem.

Proof of Theorem 1. For each tree C_i, $i = 1, \ldots, km$, the adversary de-pebbles $\mathscr{W}_i$ immediately after the first pebble is placed on C_i. No node of C_{i+1} can be pebbled

before $root^{(i)}$ is pebbled. With $f \leqslant T/s$, $N \leqslant s \leqslant \hat{s} \leqslant T/39 \log s$, the number of trees is

$$km = \left\lfloor k\frac{T}{3kN} \right\rfloor \geqslant \frac{T}{4N} \geqslant \frac{T}{s}\frac{N}{4\hat{s}} \geqslant f\frac{N}{4\hat{s}}.$$

Thus, with a total of f faults, and by Claim 4, we conclude that the adversary can de-pebble at least $fN/4\hat{s}$ times. Combining this with Claim 5 we obtain

$$\begin{aligned} T_G(\hat{s}, f) &\geqslant f\frac{N}{4\hat{s}}\left(\frac{T}{12} - 3k\hat{s}\right) + |G| \\ &\geqslant f\frac{s}{16\hat{s}}\left(\frac{T}{12} - 3\hat{s}\log s\right) + \Theta(T) = \Omega\left(\left(f\frac{s}{\hat{s}} + 1\right)T\right), \end{aligned}$$

for $\hat{s} \leqslant T/39 \log s$, implying the theorem. □

4. The upper bound

We now present an effective pebbling strategy, with a performance which matches the lower bound (up to a constant factor). In essence, the strategy calls for keeping "snap-shots" of the entire memory at fixed intervals during the computation.

Consider a graph G with $T_G(s,0) = T$. Let $\mathscr{A}$ be the pebbling strategy which achieves this performance. Suppose we are given $\hat{s} > s$ pebbles. First consider the case where $\hat{s} < T$. We use the additional $(\hat{s} - s)$ pebbles to obtain the snap-shots of the memory. Set $c = \lfloor \hat{s}/s \rfloor$ and $\tau = \lceil T/c \rceil$. For each $i = 1, \ldots, c-1$, let $N(i)$ be the set of nodes of G which, according to $\mathscr{A}$, hold a pebble at time $i \cdot \tau$. We call $N(i)$ the ith *contour*.

The strategy to pebble G in the faulty case is composed of three sub-strategies: *Compute*, *Snap-shot* and *Recompute*.

Compute: This pebbling follows the original pebbling strategy $\mathscr{A}$. It is interrupted by occasional snap-shot pebbling. In addition, in case of a failure it may also be interrupted by recomputations.

Snap-shot: After $i\tau$ steps of compute pebbling, an additional pebble is placed on each node of $N(i)$. We call this the *ith snap-shot*, denoted by $SN(i)$. For completeness, we set $SN(0) = \emptyset$. All compute pebbles placed after the ith snap-shot and before the $(i+1)$, are said to *depend* on $SN(i)$. In addition all pebbles of $SN(i+1)$ are also said to depend on $SN(i)$. Note that for all i, $|SN(i)| \leqslant s$.

Recompute: Recomputation is performed in response to a failure. Recomputation has the highest priority and interrupts any other pebbling process. If a pebble is removed (by the adversary), then this pebble is recomputed from the snap-shot on which it is dependent.

Specifically, suppose pebble p is removed, and that it depends on $SN(i)$. Pebble p is now to be recomputed. To this end we remove *all* other pebbles depending on $SN(i)$. These pebbles, together with p, are now recomputed from $SN(i)$, following the original pebbling strategy $\mathscr{A}$. The adversary can also remove several pebbles, depending

on different contours. In this case, pebbles depending on different contours are recomputed independently, and at any time, the pebbles depending on the lowest contour are recomputed first. (A fault in a lower contour interrupts the reconstruction of a higher one.) If a fault occurs in the course of the reconstruction, then the reconstruction of the contour is restarted from scratch. In this way, any single fault results in either reconstruction of a single contour, or in redoing the work in between contours. In both cases the extra work is at most τ.

This completes the description of the strategy for the case $\hat{s} < T$.

Now suppose that $\hat{s} \geqslant T$. Since $T_G(s,0) = T$ it must be that G has at most T nodes. Thus, using T pebbles we can employ the naive pebbling strategy where each node is assigned a pebble, and this pebble is not removed (by the computing player). With this strategy, each failure requires at most one additional pebbling step. We denote the union of these strategies by Snap-S($\hat{s}$).

Theorem 2. *Let G be a graph. For any s such that $T_G(s,0) < \infty$ and for any f. Let $\hat{s}$ satisfy $\hat{s} \geqslant s$, $\hat{s} = \mathrm{O}(T)$ and $\hat{s} = \mathrm{O}(sf)$. Then* Snap-S($\hat{s}$) *uses at most $\hat{s}$ pebbles, and obtains*

$$\frac{\hat{s} \cdot T_G(\hat{s}, f)}{s \cdot T_G(s,0)} = \mathrm{O}(f).$$

Proof. Set $T = T_G(s,0)$. If $\hat{s} \geqslant T$ then Snap-S($\hat{s}$) uses T pebbles, and completes the pebbling in at most $T + f$ steps. Consider the case $\hat{s} < T$. To prove the bound on the number of pebbles, partition the $\hat{s}$ pebbles into $c = \lfloor \hat{s}/s \rfloor$ sets, $A_0, \ldots, A_{c-1}$, each consisting of s pebbles. Pebbles of A_0 will be used as in the original pebbling strategy. Pebbles of A_i will be used for the ith snap-shot. For the recomputation, if a pebble of A_j is recomputed, then only pebbles of A_j are used. Since $|A_j| = s$ and s pebbles are used in the original pebbling strategy, this amount will suffice for any recomputation (recall that before recomputation, we remove *all* the pebbles of A_j from the graph). Thus, Snap-S($\hat{s}, f$) indeed uses at most $\hat{s}$ pebbles.

Now let us count the number of pebbling steps. The *compute* pebbling takes exactly T steps. The *snap-shots* take in total $(c-1)s$. Each *recomputation* takes at most τ steps, and there are at most f complete recomputations. Thus, in total,

$$T_G(\hat{s}, f) \leqslant T + (c-1)s + f\tau < T + \left\lfloor \frac{\hat{s}}{s} \right\rfloor s + f \left\lceil \frac{T}{\lfloor \hat{s}/s \rfloor} \right\rceil \leqslant 2T + 2f\frac{s}{\hat{s}}T. \qquad \square$$

Note that the strategy Snap-S($\hat{s}$) is dependent only on G and $\hat{s}$, *not* on f. Thus, it may be used uniformly, without a priori knowledge of the expected number of faults.

5. A special case: Binary trees

In this section we show that for special classes of graphs better bounds can be obtained. We exemplify this by considering the class of binary trees. Consider a tree

computation which in the fault-free case requires s pebbles. We show that, given $\hat{s}$ pebbles and faced with at most f faults, this computation can be completed with only a $\Theta(f/(\hat{s}-s))$ time overhead. Thus, the overhead is inverse in the *additive* factor of extra memory provided. This is in contrast to the general case, where the determining quantity is the *multiplicative* factor $(\hat{s}/s)$.

The following is general fact for the pebbling of trees.

Lemma 2. *Let G be a tree on n nodes and let $T_G(s,0) = t < \infty$. Then $t = n$.*

Proof. Each node in the tree needs to be pebbled at least once. We show that no node needs to be pebbled more than once. Otherwise, consider the node v closest to the root that was pebbled twice. The direct follower of this node is pebbled only once, and hence one of the two occasions on which v was pebbled could not serve any useful purpose. □

Thus, for trees we may interchange between $T_G(s,0)$ and n, whenever s pebbles suffice.

For the upper bound we use an algorithm similar to the snap-shot algorithm. In this case, however, we are more careful as to where to place the pebbles, and which pebbles to use for recomputing. For this we use the following fact:

Claim 6. *Let G be an n node binary tree. For any τ, $0 \leqslant \tau \leqslant n$ there exists a sub-tree $\widetilde{G}$ such that $\tau \leqslant |\widetilde{G}| \leqslant 2\tau + 1$.*

Proof. By induction on n. The case $n = 1$ is clear. Assume for all $n' < n$. If $\tau \geqslant (n-1)/2$ then $\widetilde{G} = G$. Otherwise, let G_1 and G_2 be the subtrees of the root (each possibly empty). It must be that at least for one $i \in \{1,2\}$, $|G_i| = n_i \geqslant (n-1)/2$. By the inductive assumption G_i contains the desired sub-tree. □

Let G be an n node binary tree. Suppose we are given $\hat{s}$ pebbles to pebble G. We will be using s pebbles to perform the computations and recomputations, and the additional $\hat{s}-s$ pebbles to hold "check points". We call the first set of pebbles the *active pebbles* and the second the *fixed pebbles*.

The algorithm works in two modes of operation: *compute* and *recompute*:

Compute: Set $\tau = n/(\hat{s}-s)$. Set $G_0 = G$. In G_0 there exists a sub-tree $\widetilde{G}_0$ with $\leqslant 2\tau + 1$ nodes. Since s pebbles suffice to compute G they surely suffice to compute $\widetilde{G}_0$. Thus, we first compute this tree with the s active pebbles, placing a fixed pebble on the root. Now, let $G_1 = G_0 - \widetilde{G}_0$, the tree remaining after removing the pebbled tree $\widetilde{G}_0$. By induction, the process is now repeated, until $|G_i| \leqslant n/(\hat{s}-s)$, in which case the entire graph is pebbled, and the pebbling complete. This pebbling process may be interrupted by recomputation phases.

Recomputation: If a pebble is removed by the adversary then it is recomputed from the previous fixed pebbles, using the active pebbles for the process. Thus, in case of a failure, the current computation is aborted, and lost. It will be redone once the recomputation is completed. If the adversary removes several pebbles then the earliest computed pebbles are recomputed first. This is done in a priority based fashion, where recomputation of one pebble may be temporarily interrupted by a need to recompute a previous pebble, due to a new fault.

Theorem 3. *Let G be a binary tree such that $T(s,0) < \infty$. For any $\hat{s} > s$, the above algorithm uses at most $\hat{s}$ pebbles, and for any f obtains*

$$(\hat{s} - s)\frac{T_G(\hat{s}, f)}{T_G(s,0)} = \mathrm{O}(f).$$

Proof. Set $T = T_G(s,0)$. Each time a fixed pebble is placed at least $\tau = n/(\hat{s} - s)$ nodes are removed from the graph. Thus, at most $(\hat{s} - s)$ fixed pebbles are placed. At most s pebbles are used for the computation and recomputations. Thus, in total, no more than $\hat{s}$ pebbles are employed.

The basic computation requires T steps. Each recomputation requires at most $2\tau + 1$ steps. In addition, the recomputation may necessitate to re-do a computation of at most $2\tau + 1$ steps. Thus, a total f faults results in at most $4\tau f + 2f = f4n/(\hat{s} - s) + 2f$ extra steps. □

We can also prove a matching lower bound. We use the following lemma.

Lemma 3. *Consider a pebbling process for a full binary tree of depth d. As long as there exists a leaf which has not yet been pebbled, $d + 2$ pebbles are necessary and sufficient in order to pebble the root* (*counting also pebbles already located on the tree*).

Proof. The proof proceeds by induction on d. For $d = 1$, three pebbles are necessary. Assume for $d - 1$, we prove for d. Let G_0 and G_1 be the left and right subtrees of the root, and let r_0 and r_1 be their roots, respectively. W.l.o.g. assume that the unpebbled leaf is in G_0. In order to pebble the root, first pebble r_0 using $d + 1$ pebbles (by the inductive hypothesis), leave a pebble on node $d - 1$, then use $d + 1$ pebbles in order to pebble r_1, and then reuse one of these pebbles in order to pebble the root. To see that $d + 2$ pebbles are necessary, observe that both r_0 and r_1 need to be pebbled. By the induction hypothesis, pebbling r_0 requires $d + 1$ pebbles. If at the time step that $d + 1$ pebbles are used for this purpose, G_1 contains some pebble, then $d + 2$ pebbles have been used. Otherwise, this other subtree itself needs $d + 1$ pebbles in order to be pebbled. This implies that either progress cannot be made, because whenever one subtree contains $d + 1$ pebbles the other contains none, or $d + 2$ pebbles are used. □

Theorem 4. *For any T and s, $s < \log T$, there exists a binary tree $G = G_{T,s}$ such that $T_G(s,0) = T$ and for all f and $\hat{s} > s$,*

$$(\hat{s} - s)\frac{T_G(\hat{s}, f)}{T_G(s,0)} = \Omega(f).$$

Proof. We prove the theorem for $s \geqslant 5$. Consider a full binary tree of depth $s-3$ (containing 2^{s-3} leaves). By Lemma 3, $s-1$ pebbles are necessary and sufficient in order to pebble it. Any pebbling scheme for this tree must reach a *critical* time step in which $s-2$ pebbles suffice in order to complete the pebbling. This critical time step must occur only after all leaves have been pebbled (by Lemma 3).

Consider now a full binary tree of depth $s-2$, composed of a root joining a left subtree and a right subtree, each of depth $s-3$. Any pebbling process for this tree that uses $\hat{s} > s$ pebbles must reach a critical time steps for each of the two subtrees. Assume w.l.o.g., that the critical time step for the left subtree was reached first. Then at the critical time step of the right subtree, the adversary removes all pebbles from the left subtree, a total of at most $\hat{s}-(s-2)$ pebbles. The amount of work lost for the pebbling algorithm is at least 2^{s-3}, since all leaves of the left subtree must be repebbled. The adversary repeats this strategy of waiting until the pebbling process reaches a critical step for one subtree, and depebbling the other subtree (which had previously reached its critical time step). The number of repetitions is at least $\lfloor f/(\hat{s}-(s-2)) \rfloor$, and the pebbling time is at least $(2^{s-1}-1) + 2^{s-3}\lfloor f/(\hat{s}-(s-2)) \rfloor$.

This proves our theorem for the special case that $T = (2^{s-1}-1)$. For larger values of T, replace the leftmost leaf and the rightmost leaf of the full binary tree by two chains of (approximately) equal length, completing the number of nodes to T. In each repebbling event, one of the two chains must be repebbled. □

References

[1] S. Cook, An observation on time-storage trade-off, JCSS 9 (1974) 308–316.
[2] C.E. Hewitt, M.S. Paterson, Comparative schematology, Project MAC Conf. on Concurrent Systems and Parallel Computation, Woods Hole, MA, 1970, pp. 119–127.
[3] J. Hopcroft, W. Paul, L. Valiant, On time versus space, JACM 24(2) (1977) 332–337.
[4] T. Lengauer, R.E. Tarjan, Asymptotically tight bounds on time-space trade-offs in a pebble game, J. ACM 29 (1982) 1087–1130.
[5] J. von Neumann, O. Morgenstern, Theory of Games and Economic Behavior, Princeton University Press, Princeton, 1944.
[6] M.O. Rabin, Efficient dispersal of information for security, load balancing and fault tolerance, JACM 36(2) (1989) 335–348.
[7] M.O. Rabin, The Information Dispersal Algorithm and its Applications, in: R.M. Capocelli (Ed.), Sequences, Springer, Berlin, 1990, pp. 406–419.
[8] R. Sethi, Complete Register Allocation Problems, SIAM J. Comput. 4 (1975) 226–248.